MO401

Arquitetura de Computadores I

2006

Prof. Paulo Cesar Centoducatte
ducatte@ic.unicamp.br
www.ic.unicamp.br/~ducatte

MO401

Arquitetura de Computadores I

Paralelismo em Nível de Instruções
Exploração Estática

"Computer Architecture: A Quantitative Approach" - (Capítulo 4)

Paralelismo em Nível de Instruções
Exploração Estática

1. Técnicas de Compilação para Explorar ILP
2. Static Branch Prediction
3. Múltiplos Issue Estático: VLIW
4. Suporte Avançados à Compilação para ILP
5. Suporte de Hardware para Expor mais Paralelismo

Técnicas para redução de stalls

	Technique	Reduces
Capítulo 3	Dynamic scheduling	Data hazard stalls
	Dynamic branch prediction	Control stalls
	Issuing multiple instructions per cycle	Ideal CPI
	Speculation	Data and control stalls
	Dynamic memory disambiguation	Data hazard stalls involving memory
Capítulo 4	Loop unrolling	Control hazard stalls
	Basic compiler pipeline scheduling	Data hazard stalls
	Compiler dependence analysis	Ideal CPI and data hazard stalls
	Software pipelining and trace scheduling	Ideal CPI and data hazard stalls
	Compiler speculation	Ideal CPI, data and control stalls

Paralelismo em Nível de Instruções
Exploração Estática

- Até agora exploramos ILP em HW: Reservation Stations, ROB, BTB, Regs. Virtuais ...
- Como o compilador pode ajudar a melhorar o desempenho?
  - reconhecendo e tirando vantagens de ILP, como?
    - » Analisando o código e aplicando transformações que, preservando a semântica, exponha mais paraleleismo ILP.

Static Branch Prediction

- Solução mais simples: Predict Taken
  - average misprediction rate = freqüência de branches não tomados, que para os programas do SPEC é 34%.
  - misprediction rate varia de 59% a 9%
- Predição baseada na direção do branch?
  - backward-going branches: taken -> loop
  - forward-going branches: not taken -> if?
  - Programas SPEC: maioria dos forward-going branches são tomados => taken é melhor
- Predição baseada em informações de profile (coletadas em execuções anteriores.
  - Misprediction varia de 5% a 22%

Exemplo

Considere o seguinte código:

```
for (i=1000; i>0; i=i-1)
    x[i] = x[i] + s;
```

Assuma as seguinte latências para os exemplos a seguir:

Instr. produzindo resultado	*Instrução usando resultado*	*Execução em ciclos*	*Latência em ciclos*
FP ALU op	Another FP ALU op	4	3
FP ALU op	Store double	3	2
Load double	FP ALU op	1	1
Load double	Store double	1	0
Integer op	Integer op	1	0

Forwarding

Pipeline Scheduling e Loop Unrolling

Exemplo: seja o loop abaixo

```
for (i=1; i<=1000; i++)
    x(i) = x(i) + s;
```

E o assembler equivalente

```
Loop: LD    F0, 0(R1)  ;F0 = elemento do vetor - x(i)
      ADDD  F4, F0, F2 ;add escalar em F2
      SD    0(R1), F4  ;armazena o resultado
      SUBI  R1, R1, 8  ;decrementa o pointer 8bytes (DW)
      BNEZ  R1, Loop   ;branch R1!=zero
      NOP              ;delayed branch slot
```

Onde estão os stalls?

FP Loop Hazards - MIPS

```
Loop: LD    F0,0(R1)  ;F0=elemento do vetor - x(i)
      ADDD  F4,F0,F2  ;add escalar  em F2
      SD    0(R1),F4  ;armazena o resultado
      SUBI  R1,R1,8   ;decrementa o pointer 8bytes (DW)
      BNEZ  R1,Loop   ;branch R1!=zero
      NOP             ;delayed branch slot
```

Assuma as seguintes latências:

Instr. produzindo resultado	*Instrução usando resultado*	*Latência em ciclos de clock*
FP ALU op	Another FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1
Load double	Store double	0
Integer op	Integer op	0

FP Loop - Stalls

```
1  Loop: LD    F0, 0(R1)   ;F0=elemento do vetor
2        stall
3        ADDD  F4, F0, F2  ;add escalar em F2
4        stall
5        stall
6        SD    0(R1), F4   ;armazena o resultado
7        SUBI  R1, R1, 8   ;decrementa o pointer 8Byte (DW)
8        stall
9        BNEZ  R1, Loop    ;branch R1!=zero
10       stall             ;delayed branch slot
```

Instr. produzindo resultado	*Instrução usando resultado*	*Latência em ciclos de clock*
FP ALU op	Another FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1
Load double	Store double	0
Integer op	Integer op	0

10 clocks: reescreva o código minimizando os stalls?

FP Loop: Minimizando Stalls Refazendo o Escalonamento

```
1 Loop: LD    F0, 0(R1)
2       SUBI  R1, R1, 8
3       ADDD  F4, F0, F2
4       stall
5       BNEZ  R1, Loop ;delayed branch
6       SD    8(R1), F4 ; R1 alterado por SUBI
```

Trocar a ordem das instruções e ajustar o endereço de SD

Latências para Operações FP

Instr. produzindo resultado	*Instrução usando resultado*	*Latência em ciclos de clock*
FP ALU op	Another FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1
Load double	Store Double	0

Agora: 6 clocks. Como melhorar ainda mais?
Loop Unrolling

Loop unrolling: minimizando os Stalls

```
1  Loop: L.D    F0,0(R1)
2        ADD.D  F4,F0,F2
3        S.D    0(R1),F4    ;Sem DSUBUI & BNEZ
4        L.D    F6,-8(R1)
5        ADD.D  F8,F6,F2
6        S.D    -8(R1),F8   ; Sem DSUBUI & BNEZ
7        L.D    F10,-16(R1)
8        ADD.D  F12,F10,F2
9        S.D    -16(R1),F12 ; Sem DSUBUI & BNEZ
10       L.D    F14,-24(R1)
11       ADD.D  F16,F14,F2
12       S.D    -24(R1),F16
13       DSUBUI R1,R1,#32   ;alterado para 4*8
14       BNEZ   R1,LOOP
15       NOP
```

1 cycle stall
2 cycles stall

Rewrite loop to minimize stalls?

15 + 4 x (1+2) = 27 clock cycles, ou 6.8 por iteração
Assumindo R1 como múltiplo de 4

Aplicando Loop Unrolling: 4 vezes

```
1 Loop:  LD     F0,0(R1)
2        stall
3        ADDD   F4,F0,F2
4        stall
5        stall
6        SD     0(R1),F4
7        LD     F6,-8(R1)
8        stall
9        ADDD   F8,F6,F2
10       stall
11       stall
12       SD     -8(R1),F8
13       LD     F10,-16(R1)
14       stall
```

```
15       ADDD   F12,F10,F2
16       stall
17       stall
18       SD     -16(R1),F12
19       LD     F14,-24(R1)
20       stall
21       ADDD   F16,F14,F2
22       stall
23       stall
24       SD     -24(R1),F16
25       SUBI   R1,R1,#32
26       BNEZ   R1,LOOP
27       stall
28       NOP
```

15 + 4 x (1+2) +1 = 28 ciclos de clock, ou 7 por iteração. (- 3 branches e 3 SUBI)

Assumindo que R1 é múltiplo de 4, reescreva o loop minimizando os Stalls.

Loop unrolling: minimizando os Stalls

```
1 Loop:  LD     F0,0(R1)
2        LD     F6,-8(R1)
3        LD     F10,-16(R1)
4        LD     F14,-24(R1)
5        ADDD   F4,F0,F2
6        ADDD   F8,F6,F2
7        ADDD   F12,F10,F2
8        ADDD   F16,F14,F2
9        SD     0(R1),F4
10       SD     -8(R1),F8
11       SD     -16(R1),F12
12       SUBI   R1,R1,#32
13       BNEZ   R1,LOOP
14       SD     8(R1),F16     ; 8-32 = -24
```

O que foi feito

- Store após SUBI, com alteração no reg.
- Loads antes dos Stores: pega os dados logo.
- Quando isso é factivel pelo compilador?

Sem Stalls

14 ciclos de clock ou 3.5 por iteração

Loop Unrolling: O que fazer? (ou o que foi feito no exemplo)

- Determinar se é possível mover SD para após o SUBI e BNEZ e calcular o ajuste do offset de SD.
- Determinar se desdobrar o loop será útil avaliando-se se não há dependência entre iterações do loop, exceto para o controle do loop.
- Usar registradores diferentes, evitando restrições desnecessárias forçadas pelo uso do mesmo registrador para operações independentes.
- Eliminar os testes extras e branches e ajustes no código de controle do loop.
- Determinar se os loads e stores podem ser, no loop desdobrado, trocados de lugar baseado em que loads e stores de iterações diferentes são independentes. Isto requer uma análise de endereçamento de memória e determinar que eles não são o mesmo endereço.
- Escalonar o código preservando todas as dependências necessárias a se manter a semântica do código original.

Loop Unrolling: n Iterações, Como Tratar?

- Em geral não conhecemos a priori o número de iterações e nem mesmo um limite superior para ele
- Suponha que o número de iterações seja n e queremos desdobrar o loop em k cópias do seu corpo
- No lugar de gerarmos um único loop (novo) desdobrado geramos um par de loops consecutivos:
  - 1º loop: executa (n mod k) vezes e tem o corpo do loop original
  - 2º loop: é um loop com o corpo original desdobrado em K e que executa (n/k) vezes (divisão inteira)
  - Para valores grandes de n a maior parte do tempo de execução é gasta no loop desdobrado.

Compilador: Movimentação de Código

- O Compilador deve focar nas dependências existentes no programa e não se os hazards dependem de um dado pipeline
- Tentar produzir um escalonamento que evite os hazards que reduzem o desempenho
- (True) Data dependencies (RAW)
  - Instrução i produz um resultado usado pela instrução j, ou
  - Instrução j é dependente de dados da instrução k e a instrução k é dependente de dados da instrução i.
- Se dependente, não podem ser executadas em paralelo
- Fácil de ser determinado em registradores (nomes "únicos")
- Difícil para memória (problema denominado "memory disambiguation"):
  - Ex.: 100(R4) = 20(R6)?
  - E em diferentes iterações do loop: 20(R6) = 20(R6) e 100(R4) = 20(R6)?

Compilador: Movimentação de Código Dependência de Dados

- Aonde está a dependência de Dados?

```
1 Loop:  LD     F0, 0(R1)
2        ADDD   F4, F0, F2
3        SUBI   R1, R1, 8
4        BNEZ   R1, Loop    ;delayed branch
5        SD     8(R1), F4   ;alterado pelo movimento de SUBI
```

Compilador: Movimentação de Código
Dependência de Nome

- Name Dependence:
  duas instruções usam o mesmo nome (registrador ou memória) mas não compartilham dados
- Anti-dependence (WAR se há um hazard no HW)
  - Instrução j escreve em um registrador (ou posição de memória) que é lido pela instrução i e i é executado primeiro
- Output dependence (WAW se há um hazard no HW)
  - Instruções i e j escrevem no mesmo registrador (ou posição de memória); a ordem das instruções deve ser preservada.

Compilador: Movimentação de Código
Dependência de Nome.
Aonde esta?

```
1 Loop: LD    F0,0(R1)
2       ADDD  F4,F0,F2
3       SD    0(R1),F4
4       LD    F0,-8(R1)
5       ADDD  F4,F0,F2
6       SD    -8(R1),F4
7       LD    F0,-16(R1)
8       ADDD  F4,F0,F2
9       SD    -16(R1),F4
10      LD    F0,-24(R1)
11      ADDD  F4,F0,F2
12      SD    -24(R1),F4
13      SUBI  R1,R1,#32
14      BNEZ  R1,LOOP
15      NOP
```

Nenhum dado é passado por F0, porém F0 não pode ser reusado no ciclo 4.

Como remover essa dependência?

Compilador: Movimentação de Código
Dependência de Nome.
Aonde esta?

```
1 Loop: LD    F0, 0(R1)
2       ADDD  F4, F0, F2
3       SD    0(R1), F4
4       LD    F6, -8(R1)
5       ADDD  F8, F6, F2
6       SD    -8(R1), F8
7       LD    F10, -16(R1)
8       ADDD  F12, F10, F2
9       SD    -16(R1), F12
10      LD    F14, -24(R1)
11      ADDD  F16, F14, F2
12      SD    -24(R1), F16
13      SUBI  R1, R1, #32
14      BNEZ  R1, LOOP
15      NOP
```

Agora só existe dependência de dados.

"register renaming"

Compilador: Movimentação de Código
Dependência de Nome.

- Dependência de Nome é difícil de ser determinada para acessos à memória
  - 100(R4) = 20(R6)?
  - Para iterações diferentes do loop, 20(R6) = 20(R6)?
- No exemplo é necessário que o compilador determine que se R1 não é alterado então:

  $0(R1) \neq -8(R1) \neq -16(R1) \neq -24(R1)$

  e não existem dependências entre os loads e stores e assim a ordem de execução entre eles pode ser alterada

Compilador: Movimentação de Código
Dependência de Controle

- Exemplo

```
if p1 {S1;};
if p2 {S2;};
```

S1 é dependente de controle de p1 e S2 é dependente de controle de p2, mas não de p1.

Compilador: Movimentação de Código
Dependência de Controle

- Duas restrições devido a dependência de controle:
  - Uma instrução dependente de controle de um branch não pode ser movido para antes do branch, pois sua execução deixaria de ser controlada por ele.
  - Uma instrução que não é dependente de controle de um branch não pode ser movida para depois do branch, pois sua execução passaria a ser controlada por ele.
- Pode-se relaxar a dependência de controle para ter mais paralelismo, porém deve-se preservar o efeito da ordem de exceções e o fluxo de dados (manter a semântica).

Compilador: Movimentação de Código
Dependência de Controle

```
1 Loop:  LD     F0 0(R1)
2        ADDD   F4, F0, F2
3        SD     0(R1), F4
4        SUBI   R1, R1, 8
5        BEQZ   R1, exit
6        LD     F0, 0(R1)
7        ADDD   F4, F0, F2
8        SD     0(R1), F4
9        SUBI   R1, R1, 8
10       BEQZ   R1, exit
11       LD     F0, 0(R1)
12       ADDD   F4, F0, F2
13       SD     0(R1), F4
14       SUBI   R1, R1, 8
15       BEQZ   R1, exit
....
```

Aonde estão as dependências de controle?

Paralelismo em Loops
loop unrolling

- Exemplo: Aonde estão as dependências de dados? (A,B,C distintos & sem overlapping)

```
for (i=1; i<=100; i=i+1) {
        A[i+1] = A[i] + C[i];     /* S1 */
        B[i+1] = B[i] + A[i+1];   /* S2 */
}
```

1. S2 usa o valor, A[i+1], computado por S1 na mesma iteração.
2. S1 usa um valor computado por S1 na iteração anterior, logo iteração i computa A[i+1] que é lido na iteração i+1. O mesmo ocorre com S2 para B[i] e B[i+1].

   Isto é denominado "loop-carried dependence". São dependências entre iterações

- Implica que as iterações são dependentes, e não podem ser executadas em paralelo, certo??
- Note, para o exemplo, que as iterações são distintas.

Loop-Carried Dependence
Não há paralelismo?

Considere:

```
for (i=0; i< 8; i=i+1) {
      A = A + C[i];     /* S1 */
}
```

E a Computação:

```
"Ciclo 1": temp0 = C[0] + C[1];
           temp1 = C[2] + C[3];
           temp2 = C[4] + C[5];
           temp3 = C[6] + C[7];
"Ciclo 2": temp4 = temp0 + temp1;
           temp5 = temp2 + temp3;
"Ciclo 3":   A = temp4 + temp5;
```

Possível devido a natureza associativa da "+".

Paralelismo em Loops
loop unrolling

- Exemplo: Aonde estão as dependências de dados? (A,B,C distintos & sem overlapping)

```
for (i=1; i<=100; i=i+1) {
        A[i+1] = A[i] + B[i];     /* S1 */
        B[i+1] = C[i] + D[i];     /* S2 */
}
```

1. Não há dependência entre S1 e S2. Se houver, então será uma dependência ciclica e o loop não poderá ser paralelizável. Como não há dependência então pode-se trocar a ordem de execução das sentênças sem afetar o resultado de S2

2. Na primeira iteração do loop, a sentença S1 depende do valor B[1] computado antes da iniciar a execução do loop.

Paralelismo em Loops
loop unrolling

```
for (i=1; i<=100; i=i+1) {
    A[i] = A[i] + B[i];      /* S1 */
    B[i+1] = C[i] + D[i];}   /* S2 */
```

Sem dependências circulares.

Loop causa dependência em B.

Original:

Modificado:

```
A[1] = A[1] + B[1];
for (i=1; i<=99; i=i+1) {
    B[i+1] = C[i] + D[i];
    A[i+1] = A[i+1] + B[i+1];
}
B[101] = C[100] + D[100];
```

Eliminada a dependência de loop.

Loop Unrolling em Superscalar
Múltiplos Issue

SuperScalar Versão do MIPS

	Integer instruction	*FP instruction*	*Clock cycle*
Loop:	LD F0,0(R1)		1
	LD F6,-8(R1)		2
	LD F10,-16(R1)	ADDD F4,F0,F2	3
	LD F14,-24(R1)	ADDD F8,F6,F2	4
	LD F18,-32(R1)	ADDD F12,F10,F2	5
	SD 0(R1),F4	ADDD F16,F14,F2	6
	SD -8(R1),F8	ADDD F20,F18,F2	7
	SD -16(R1),F12		8
	SD -24(R1),F16		9
	SUBI R1,R1,#40		10
	BNEZ R1,LOOP		11
	SD 8(R1),F20		12

- Desenrolado 5 vezes para evitar delays
- 12 clocks, ou 2.4 clocks por iteração

Loop Unrolling em Superscalar Dinâmico
Múltiplos Issue

Múltiplas Instruções Issue & Scheduling Dinâmico

Iteration no.	Instructions	Issues	Executes	Writes result clock-cycle number
1	LD F0,0(R1)	1	2	4
1	ADDD F4,F0,F2	1	5	8
1	SD 0(R1),F4	2	9	
1	SUBI R1,R1,#8	3	4	5
1	BNEZ R1,LOOP	4	5	
2	LD F0,0(R1)	5	6	8
2	ADDD F4,F0,F2	5	9	12
2	SD 0(R1),F4	6	13	
2	SUBI R1,R1,#8	7	8	9
2	BNEZ R1,LOOP	8	9	

- 4 clocks por iteração

Branches, Decrementos gastam 1 ciclo de clock

Loop Unrolling em VLIW
Múltiplos Issue

Memory reference 1	Memory reference 2	FP operation 1	FP op. 2	Int. op/ branch	Clock
LD F0,0(R1)	LD F6,-8(R1)				1
LD F10,-16(R1)	LD F14,-24(R1)				2
LD F18,-32(R1)	LD F22,-40(R1)	ADDD F4,F0,F2	ADDD F8,F6,F2		3
LD F26,-48(R1)		ADDD F12,F10,F2	ADDD F16,F14,F2		4
		ADDD F20,F18,F2	ADDD F24,F22,F2		5
SD 0(R1),F4	SD -8(R1),F8	ADDD F28,F26,F2			6
SD -16(R1),F12	SD -24(R1),F16				7
SD -32(R1),F20	SD -40(R1),F24			SUBI R1,R1,#48	8
SD -0(R1),F28				BNEZ R1,LOOP	9

· Desenrolado 7 vezes para evitar delays

· 7 resultados em 9 clocks, ou 1.3 clocks por iteração

· É necessário mais registradores para uso efetivo do VLIW

Compilers and ILP

Example 1

There are NO dependencies

Loop Level Parallelism

```
/* ******************************************************
   This is the example on page 305 of Hennessy &
   Patterson but running on an Intel Machine
****************************************************** */

#define         MAX         1000
#define         ITER        100000
int         main( int argc, char argv[] )
{
    double      x[MAX + 2];
    double      s = 3.14159;
    int         i, j;

    for ( i = MAX; i > 0; i-- )     /* Init array */
        x[i] = 0;

    for ( j = ITER; j > 0; j-- )
        for ( i = MAX; i > 0; i-- )
            x[i] = x[i] + s;
}
```


Compilers and ILP

Loop Level Parallelism

Example 1

Elapsed seconds = 0.590026

```
This is the GCC optimized
  code


.L15:
        fldl  (%ecx,%eax)
        fadd %st(1),%st
        decl %edx
        fstpl (%ecx,%eax)
        addl $-8,%eax
        testl %edx,%edx
        jg .L15
```

Elapsed seconds = 0.122848

```
This is the ICC optimized code
.L2:
        fstpl     8(%esp,%edx,8)
        fldl      (%esp,%edx,8)
        fadd      %st(1), %st
        fldl      -8(%esp,%edx,8)
        fldl      -16(%esp,%edx,8)
        fldl      -24(%esp,%edx,8)
        fldl      -32(%esp,%edx,8)
        fxch      %st(4)
        fstpl     (%esp,%edx,8)
        fxch      %st(2)
        fadd      %st(4), %st
        fstpl     -8(%esp,%edx,8)
        fadd      %st(3), %st
        fstpl     -16(%esp,%edx,8)
        fadd      %st(2), %st
        fstpl     -24(%esp,%edx,8)
        fadd      %st(1), %st
        addl      $-5, %edx
        testl     %edx, %edx
        jg        .L2           # Prob 99%
        fstpl     8(%esp,%edx,8)
```


Compilers and ILP

Example 2

Loop Level Parallelism

```
// Example on Page 320
    get_current_time( &start_time );
    for ( j = ITER; j > 0; j-- )
    {
        for ( i = 1; i <= MAX; i++ )
        {
            A[i+1] = A[i] + C[i];
            B[i+1] = B[i] + A[i+1];
        }
    }
    get_current_time( &end_time );
```

There are two dependencies here – what are they?

Compilers and ILP

Loop Level Parallelism

Example 2

Elapsed seconds = 1.357084

```
This is GCC optimized code
.L55:
   fldl -8(%esi,%eax)
   faddl -8(%edi,%eax)
   fstl (%esi,%eax)
   faddl -8(%ecx,%eax)
   incl %edx
   fstpl (%ecx,%eax)
   addl $8,%eax
   cmpl $1000,%edx
   jle .L55
```

```
This is Microsoft optimized code
$L1225:
    fld   QWORD PTR _C$[esp+eax+40108]
    add   eax, 8
    cmp   eax, 7992
    fadd  QWORD PTR _A$[esp+eax+40100]
    fst   QWORD PTR _A$[esp+eax+40108]
    fadd  QWORD PTR _B$[esp+eax+40100]
    fstp  QWORD PTR _B$[esp+eax+40108]
    jle   $L1225
```

Elapsed seconds = 0.664073

```
This is the ICC optimized code
.L4:
        fstpl     25368(%esp,%edx,8)
        fldl      8472(%esp,%edx,8)
        faddl     16920(%esp,%edx,8)
        fldl      25368(%esp,%edx,8)
        fldl      16928(%esp,%edx,8)
        fxch      %st(2)
        fstl      8480(%esp,%edx,8)
        fadd      %st, %st(1)
        fxch      %st(1)
        fstl      25376(%esp,%edx,8)
        fxch      %st(2)
        faddp     %st, %st(1)
        fstl      8488(%esp,%edx,8)
        faddp     %st, %st(1)
        addl      $2, %edx
        cmpl      $1000, %edx
        jle       .L4           # Prob 99%
        fstpl     25368(%esp,%edx,8)
```

Example 3

Loop Level Parallelism

```
// Example on Page 321
    get_current_time( &start_time );

    for ( j = ITER; j > 0; j-- )
    {
        for ( i = 1; i <= MAX; i++ )
        {
            A[i]   = A[i] + B[i];
            B[i+1] = C[i] + D[i];
        }
    }
    get_current_time( &end_time );
```

What are the depend-encies here??

Loop Level Parallelism

Example 3

Elapsed seconds = 1.370478

```
This is the GCC optimized code


.L65:
   fldl  (%esi,%eax)
   faddl  (%ecx,%eax)
   fstpl  (%esi,%eax)
   movl  -40100(%ebp),%edi
   fldl  (%edi,%eax)
   movl  -40136(%ebp),%edi
   faddl  (%edi,%eax)
   incl %edx
   fstpl 8(%ecx,%eax)
   addl $8,%eax
   cmpl $1000,%edx
   jle .L65
```

Elapsed seconds = 0.325419

```
This is the ICC optimized code
 .L6:
        fstpl     8464(%esp,%edx,8)
        fldl      8472(%esp,%edx,8)
        faddl     25368(%esp,%edx,8
        fldl      16920(%esp,%edx,8)
        faddl     33824(%esp,%edx,8)
        fldl      8480(%esp,%edx,8)
        fldl      16928(%esp,%edx,8)
        faddl     33832(%esp,%edx,8)
        fxch      %st(3)
        fstpl     8472(%esp,%edx,8)
        fxch      %st(1)
        fstl      25376(%esp,%edx,8)
        fxch      %st(2)
        fstpl     25384(%esp,%edx,8)
        faddp     %st, %st(1)
        addl      $2, %edx
        cmpl      $1000, %edx
        jle       .L6            # Prob 99%
        fstpl     8464(%esp,%edx,8)
```


Example 4

Loop Level Parallelism

```
// Example on Page 322
    get_current_time( &start_time );
    for ( j = ITER; j > 0; j-- )
    {
        A[1]   = A[1] + B[1];
        for ( i = 1; i <= MAX - 1; i++ )
        {
            B[i+1] = C[i] + D[i];
            A[i+1] = A[i+1] + B[i+1];
        }
        B[101]   = C[100] + D[100];
    }
    get_current_time( &end_time );
```

Elapsed seconds = 1.200525

How many depend-encies here??

Loop Level Parallelism

Example 4

Elapsed seconds = 1.200525

```
This is the GCC optimized code


.L75:
   movl -40136(%ebp),%edi
   fldl -8(%edi,%eax)
   faddl -8(%esi,%eax)
   movl -40104(%ebp),%edi
   fstl (%edi,%eax)
   faddl  (%ecx,%eax)
   incl %edx
   fstpl  (%ecx,%eax)
   addl $8,%eax
   cmpl $999,%edx
   jle .L75
```

```
This is the Microsoft optimized code

$L1239
   fld   QWORD PTR _D$[esp+eax+40108]
   add   eax, 8
   cmp   eax, 7984         ;00001f30H
   fadd  QWORD PTR _C$[esp+eax+40100]
   fst   QWORD PTR _B$[esp+eax+40108]

   fadd  QWORD PTR _A$[esp+eax+40108]
   fstp  QWORD PTR _A$[esp+eax+40108]
   jle   SHORT $L1239
```


Loop Level Parallelism

Example 4

Elapsed seconds = 0.359232

```
This is the ICC optimized code


.L8:
        fstpl     8472(%esp,%edx,8)
        fldl      16920(%esp,%edx,8)
        faddl     33824(%esp,%edx,8)
        fldl      8480(%esp,%edx,8)
        fldl      16928(%esp,%edx,8)
        faddl     33832(%esp,%edx,8)
        fldl      8488(%esp,%edx,8)
        fldl      16936(%esp,%edx,8)
        faddl     33840(%esp,%edx,8)
        fldl      8496(%esp,%edx,8)
        fxch      %st(5)
```

CONTINUED

```
        fstl      25376(%esp,%edx,8)
        fxch      %st(3)
        fstl      25384(%esp,%edx,8)
        fxch      %st(1)
        fstl      25392(%esp,%edx,8)
        fxch      %st(3)
        faddp     %st, %st(4)
        fxch      %st(3)
        fstpl     8480(%esp,%edx,8)
        faddp     %st, %st(2)
        fxch      %st(1)
        fstpl     8488(%esp,%edx,8)
        faddp     %st, %st(1)
        addl      $3, %edx
        cmpl      $999, %edx
        jle       .L8
        fstpl     8472(%esp,%edx,8)
```


Suporte de Compiladores para ILP

Como os compiladores podem ser mais espertos?

1. Produzindo um bom scheduling para o código.
2. Determinando quais loops podem conter paralelismo.
3. Eliminando dependências de nome.

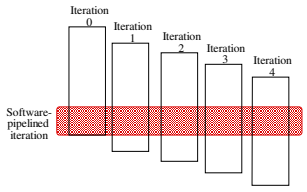
Compiladores devem ser muito espertos para se livrarem de aliases – apontadores em C são um problema.

Técnicas utilizadas:

- Symbolic Loop Unrolling
- Critical Path Scheduling

Software Pipelining
Symbolic Loop Unrolling

- Observação: se as iterações dos loops são independentes, então é possível ter mais ILP executando instruções de diferentes iterações
- Software pipelining: reorganiza o loop de forma que em cada iteração sejam executadas instruções escolhidas de diferentes iterações do loop original (~ Tomasulo em SW)



Software Pipelining
Symbolic Loop Unrolling

Exemplo: Soma dos elementos de um vetor com uma constante em F2

```
Loop:  L.D    F0,0(R1)
       ADD.D  F4,F0,F2
       S.D    0(R1),F4
       DSUBUI R1,R1,#8
       BNEZ   R1,LOOP
```

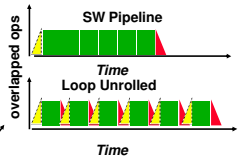

Software Pipelining: Exemplo

Antes: Desenrolar 3 vezes

```
1  L.D    F0,0(R1)
2  ADD.D  F4,F0,F2
3  S.D    0(R1),F4
4  L.D    F6,-8(R1)
5  ADD.D  F8,F6,F2
6  S.D    -8(R1),F8
7  L.D    F10,-16(R1)
8  ADD.D  F12,F10,F2
9  S.D    -16(R1),F12
10 DSUBUI R1,R1,#24
11 BNEZ   R1,LOOP
```

Depois: Software Pipelined

```
1  S.D    0(R1),F4 ; Stores M[i]
2  ADD.D  F4,F0,F2 ; Adds to M[i-1]
3  L.D    F0,-16(R1); Loads M[i-2]
4  DSUBUI R1,R1,#8
5  BNEZ   R1,LOOP
```



- Loop Unrolling Simbólico
  - Maximiza a distância resultado-uso
  - Menos código que unrolling

5 ciclos por iteração

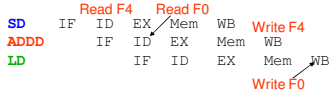
Software Pipelining: Exemplo

Antes: desenrolar 3 vezes

```
1  LD     F0,0(R1)
2  ADDD   F4,F0,F2
3  SD     0(R1),F4
4  LD     F6,-8(R1)
5  ADDD   F8,F6,F2
6  SD     -8(R1),F8
7  LD     F10,-16(R1)
8  ADDD   F12,F10,F2
9  SD     -16(R1),F12
10 SUBI   R1,R1,#24
11 BNEZ   R1,LOOP
```

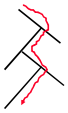
Após: Software Pipelined

```
   LD     F0,0(R1)
   ADDD   F4,F0,F2
   LD     F0,-8(R1)
1  SD     0(R1),F4;   Stores M[i]
2  ADDD   F4,F0,F2;   Adds to M[i-1]
3  LD     F0,-16(R1); loads M[i-2]
4  SUBI   R1,R1,#8
5  BNEZ   R1,LOOP
   SD     0(R1),F4
   ADDD   F4,F0,F2
   SD     -8(R1),F4
```



Trace Scheduling
Critical Path Scheduling

- Paralelismo através de IF branches vs. LOOP branches
- Dois passos:
  - *Seleção do Trace*
    - » Encontrar a(s) seqüência(s) de blocos básicos (*trace*) de maior seqüência de código (ou mais executada)
    - » Predição estática ou predição por profile
  - *Compactação do Trace*
    - » *Otimiza a execução deste código*
- *OBS.: muito usado para gerar código VLIW*



Suporte para Paralelismo

- Suporte de Software para ILP é bom quando o código é previsível em tempo de compilação.
- E se não for possível essa predição?
- Técnicas de Hardware tem que ser usadas:
  - Instruções Condicionais ou predicadas
  - Especulação em Hardware

Uso de Instruções Predicadas (Hardware e Compilador)

Suponha o seguinte código:

```
if ( VarA == 0 )
    VarS = VarT;
```

Código tradicional:

```
LD      R1, VarA
BNEZ    R1, Label
LD      R2, VarT
SD      VarS, R2
Label:
```

Compara e anula próx. Instrução se diferente de zero

Método predicado (próx.):

```
LD      R1, VarA
LD      R2, VarT
CMPNNZ  R1, #0
SD      VarS, R2
Label:
```

Compara e Move se Zero

Método predicado (instr.):

```
LD      R1, VarA
LD      R2, VarT
CMOVZ   VarS,R2, R1
```


Uso de Especulação (Hardware e Compilador)

- **Aumentando o paralelismo:**
  - A idéia é mover a execução de uma instrução através de um branch aumentando assim o tamanho do bloco básico e conseqüentemente o paralelismo.
  - Primeira dificuldade é evitar exceções. Por exemplo if (a ^= 0) c = b/a; pode causar uma divisão por zero em alguns casos.
  - Métodos para aumentar a especulação incluem:
  1. Uso de um conjunto de bits de status associados com os registradores. São um sinal que os resultados das instruções são inválidos até um certo momento.
  2. O resultado da instrução não pode ser escrito até se ter certeza que a instrução não é mais especulativa.

Uso de Especulação (Hardware e Compilador)

Suponha o seguinte código:

```
if   ( A == 0 )
     A = B;
else
     A = A + 4;
```

E assuma que A está em 0(R3) e B em 0(R2)

Código original:

```
LW     R1, 0(R3)     Load A
BNEZ   R1, L1        Testa A
LW     R1, 0(R2)     Clausula If
J      L2            Pula Else
L1:ADDI R1, R1, #4   Clausula Else
L2:SW  0(R3), R1     Store A
```

Código com especulação:

```
LW     R1, 0(R3)     Load A
LW     R14, 0(R2)    Especula Load B
BEQZ   R1, L3        Outro if Branch
ADDI   R14, R1, #4   Clausula Else
L3: SW 0(R3), R14    Non-Spec Store
```


Uso de Especulação (Hardware e Compilador)

Se no exemplo anterior o LW* produz uma exceção, então um bit de status é setado no registrador. Se uma das próximas instruções tenta usar o registrador, então uma exceção é gerada.

Código especulativo:

```
LW     R1, 0(R3)     Load A
LW*    R14, 0(R2)    Espec. Load B
BEQZ   R1, L3        Outro if Branch
ADDI   R14, R1, #4   Clausula Else
L3: SW 0(R3), R14    Não-Spec Store
```


Suporte em HW para Exceção

- Vários mecanismos garantem que a especulação realizada pelo compilador não viole o comportamento sob exceções. Por exemplo:
  - não gerar exceções em código predicado quando sua execução é anulada
  - Não gerar exceções em Prefetch

Suporte em HW para Especulação de Referências à Memória

- Para que o compilador seja capaz de mover loads através de stores, quando ele não está absolutamente certo de que esse movimento possa ser feito, instruções especiais que avaliam conflitos em endereços são necessárias e devem ser incluídas na arquitetura
  - A instrução especial é colocada na posição original do load e este é movido para antes dos stores
  - Quando o load especulativo é executado , o hardware salva o endereço do acesso à memória
  - Se um store subseqüente escreve nesta posição de memória antes que a nova instrução avalie este endereço, então a especulação falha
  - Se somente instruções de load são especuladas, então isso é suficiente para refazer o load no ponto onde a nova instrução é executada

Vantagens da Speculação em HW (Tomasulo) vs. SW (VLIW)

- HW - Vantagens:
  - Memory disambiguation: melhor em HW, uma vez que se conhece o endereço atual
  - Branch prediction : melhor em HW, possui menos overhead
  - HW mantém as exceções precisas
  - O mesmo código funciona em diferentes implementações
  - Menor código (evita muitos nops)
- SW - Vantagens:
  - A janela de instruções analisada pode ser muito maior
  - Menos hardware necessário a implementação dos VLIW
  - Mais tipos de especulação podem ser implementados
  - Especulação pode ser baseada em informações locais e globais

Superscalar vs. VLIW

- Código menor
- Compatibilidade binária através de gerações do hardware

- Hardware mais simples para decodificação e issuing das instruções
- Não possui hardware para Interlock (o compilador avalia?)
- Mais registradores, porém hardware de acesso mais simples (múltiplos e independentes register files)

Problemas com os Primeiros VLIW

- Aumento no tamanho do código
  - Geração de operações seqüenciais suficientes requer loop unrolling ambicioso
  - Quando instruções VLIW não são preenchidas, tem-se unidades funcionais não usadas (ociosas)
- Operação em lock-step; sem deteção de hazard
  - Um stall em uma unidade funcional do pipeline causa um stall na entrada do processador, já que todas as unidades funcionais trabalham sincronizadas
  - O Compilador pode predizer unidades funcionais, porém caches são dificeis de se predizer
- Compatibilidade de código binário
  - VLIW puro => diferentes números de unidades funcionais e latências necessitam diferentes códigos

Intel/HP IA-64 "Explicitly Parallel Instruction Computer (EPIC)"

- IA-64: conjunto de instruções; EPIC define o tipo
  - EPIC = 2ª geração de VLIW?
- Itanium™ é o nome da primeira implementação (2001)
  - Alto nível de paralelismo e pipeline profundo a 800 Mhz
  - 6-wide, 10-stage pipeline a 800Mhz em processo 0.18 µ
- 128 64-bit integer registers + 128 82-bit floating point registers
  - Os register files não são separado por unidade funcional como nas versões antigas de VLIW
- Hardware para avaliação de dependências (interlocks => compatibilidade binária)
- Execução predicada (seleção de 1 de 64 1-bit flags) => 40% menos mispredictions?

IA-64 Registers

- Os registradores inteiros são configurados para acelerar chamadas de procedimento usando uma pilha de registradores
  - Mecanismo similar ao desenvolvido no processador RISC-I de Berkeley e usado na arquitetura SPARC.
  - Registradores 0-31 são sempre acessíveis pelos endereços 0-31
  - Registrdores 32-128 são usados como pilha de registradores e a cada procedimento é alocada um conjunto de registradores (de 0 a 96)
  - Um novo registrador stack frame é criado para cada chamada de procedimento renomeando-se os registradores em hardware;
  - Um registrador especial chamado de current frame pointer (CFM) aponta para o conjunto de registradores que é usado pelo procedimento
- 8 64-bit Branch registers – usados para manter os endereços de destinos dos branches indiretos
- 64 1-bit registers de predição

IA-64 Registers

- Ambos os conjuntos de registradores, inteiros e ponto flutuante, suportam rotações de registradores para os registradores 32-128.
- Rotação de registradores foi projetada para facilitar a tarefa de alocação de registradores em software pipelined loops
- Quando combinado com predicação é possível evitar desrolamento e código separado de prologo e epilogo em um dado software pipelined loop
  - Torna o SW-pipelining usável para loops com um número menor de iterações

Intel/HP IA-64 "Explicitly Parallel Instruction Computer (EPIC)"

- Instruction group: uma seqüencia de instruções consecutivas sem dependência de dados nos registradores
  - Todas as instruções do grupo podem ser executadas em paralelo, se existir hardware suficiente e se as dependências em memória são preservadas
  - Um instruction group pode ter um tamanho arbitrário, mas o compilador deve explicitamente indicar os limites entre dois instruction grup colocando um stop entre as 2 instruções que pertencem a diferentes grupos
- As instruções IA-64 são codificadas em bundles de 128 bits.
  - cada bundle consiste de um campo template de 5-bit e 3 instruções, cada uma de 41 bits
  - 3 Instruções em grupos de 128 bit; os 5 bits determinam se as instruções são dependentes ou independentes
  - Código menor que os antigos VLIW, maior que o x86/RISC

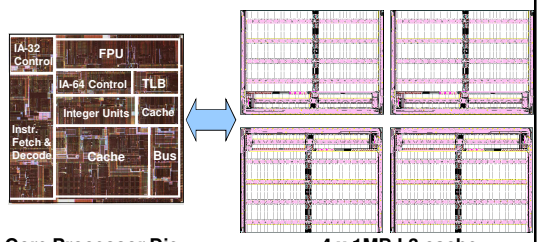
5 Tipos de Execuções em um Bundle

Execution Unit Slot	*Instruction type*	*Instruction Description*	*Example Instructions*
I-unit	A	Integer ALU	add, subtract, and, or, cmp
	I	Non-ALU Int	shifts, bit tests, moves
M-unit	A	Integer ALU	add, subtract, and, or, cmp
	M	Memory access	Loads, stores for int/FP regs
F-unit	F	Floating point	Floating point instructions
B-unit	B	Branches	Conditional branches, calls
L+X	L+X	Extended	Extended immediates, stops

- o campo de 5-bit em cada bundle descreve se há um stop associado com o bundle e o tipo da unidade de execução requerida por cada instrução no bundle

Itanium™ Processor Silicon

(Copyright: Intel at Hotchips '00)



Core Processor Die

4 x 1MB L3 cache

Itanium™ Machine Characteristics

(Copyright: Intel at Hotchips '00)

Frequency	800 MHz
Transistor Count	25.4M CPU; 295M L3
Process	0.18u CMOS, 6 metal layer
Package	Organic Land Grid Array
Machine Width	6 insts/clock (4 ALU/MM, 2 Ld/St, 2 FP, 3 Br)
Registers	14 ported 128 GR & 128 FR; 64 Predicates
Speculation	32 entry ALAT, Exception Deferral
Branch Prediction	Multilevel 4-stage Prediction Hierarchy
FP Compute Bandwidth	3.2 GFlops (DP/EP); 6.4 GFlops (SP)
Memory -> FP Bandwidth	4 DP (8 SP) operands/clock
Virtual Memory Support	64 entry ITLB, 32/96 2-level DTLB, VHPT
L2/L1 Cache	Dual ported 96K Unified & 16KD; 16KI
L2/L1 Latency	6 / 2 clocks
L3 Cache	4MB, 4-way s.a., BW of 12.8 GB/sec;
System Bus	2.1 GB/sec; 4-way Glueless MP Scalable to large (512+ proc) systems

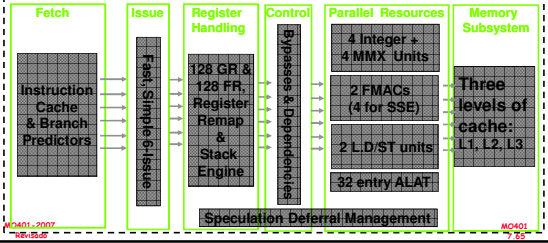
Itanium™ EPIC Design Maximizes SW-HW Synergy

(Copyright: Intel at Hotchips '00)

Architecture Features programmed by compiler:

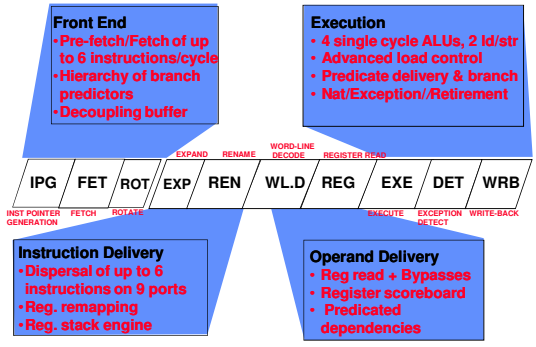
Branch Hints | *Explicit Parallelism* | *Register Stack & Rotation* | *Predication* | *Data & Control Speculation* | *Memory Hints*

Micro-architecture Features in hardware:



10 Stage In-Order Core Pipeline

(Copyright: Intel at Hotchips '00)



Processador Itanium: pipeline de 10 estágios

- Front-end (estágios IPG, Fetch e Rotate): prefetches até 32 bytes por clock (2 bundles) em um prefetch buffer, que pode manter até 8 bundles (24 instruções)
  - Branch prediction – realizado usando um multilevel adaptive predictor, como na micro-arquitetura P6
- Instruction delivery (estágios EXP e REN): distribui até 6 instruções para as 9 unidades funcionais
  - Implementa registers renaming.

Processador Itanium: pipeline de 10 estágios

- Operand delivery (WLD and REG): acessa o register file, executa register bypassing, acessa e atualiza um register scoreboard, e avalia as dependências predicadas.
  - Scoreboard usado para detectar quando uma instrução individualmente pode continuar fazendo com que um stall de uma instrução em um bundle não precise parar todo o bundle
- Execution (EXE, DET, and WRB): executa as instruções nas ALUs e nas unidades de load/store, detecta exceções e posta NaTs, executa write-back
  - O tratamento de exceções para instruções especulativas é suportado provendo NaTs (Not a Thing), equivalentes aos poison bits para os GPRs (o que torna os GPRs registradores de 65 bits), e NaT Val (Not a Thing Value) para os FPRs (já com 82 bits)

Comentários sobre o Itanium

- O Itanium tem diversas características comumente associadas com dynamically-scheduled pipelines
  - Forte enfase em branch prediction, register renaming, scoreboarding, pipeline profundo com muitos estágios antes da execução e vários estágios após a execução
  - Surpreende, para uma abordagem que a idéia principal é baseada na tecnologia de compiladores e HW simples, se ver um processador tão complexo quanto aqueles baseados em dynamically scheduled!